Common File User's Guide*

The NPARC Alliance

NASA Glenn Research Center
Cleveland, Ohio

USAF Arnold Engineering Development Center
Tullahoma, Tennessee

*This is an unnumbered version of this document, created December 5, 2007. It is essentially identical to the Common File Programmer's Guide by the Boeing Corporation. Please send corrections, additions, ideas, etc., to Mark Fisher at *mark.s.fisher@boeing.com* or Charlie Towne at *towne@grc.nasa.gov*.

Contents

1 General Overview

The *common file library* is a set of routines that provide access to a file structure hereafter called the *common file*. The common file structure is a third generation attempt to provide a unifying file structure for storing CFD data. It provides a flexible and extendable mechanism for storing CFD and other data as well. The purpose of the common file library is to insulate the applications programmer from the nuances of dealing with the myriad of different computer types that make up a computing system. By using the library the application programmer can process a common file from another machine without having to perform explicit conversions.

2 Common File Versions

The common file has evolved over the years to meet the changing needs by applications such as CFD. The current version is version 3. The following summarizes the changes to the common file for the each new version.

2.1 Version 1

Initial release of the common file with the built in limits shown in Section 7, "Default Library Values and Limits," on p. 53.

2.2 Version 2

The second release increased the limits on the number of nodes and the number of variables. This caused a minor change in how some of the internal pointers were stored. For files created with limits within version 1 there was no change in structure. Thus they could be read by version 1 libraries.

2.3 Version 3

Version 3 represents a major change in the file structure based on the CGNS (CFD General Notation System) data structure. The CGNS project was sponsored by NASA for Boeing to develop the next generation common file structure. The new structure removes the current common file limits, adds support for more data types and includes both a C and Fortran interface. The result is the ADF (Advanced Database Format) library which is written in C and supersedes the Boeing common file library. In order to maintain compatibility with old common files and provide support for the new format the ADF core has been layered under the common file interface creating version 3. Several new functions have been added as well as many limits being removed with no changes to the user interface for existing functions. Note that version 3 files cannot be read with version 1 or 2 libraries.

3 Basic Concepts

The file structure is based on a tree topology with the root node at the top and user created subnodes underneath. The node itself contains integer, real, and character data as well as pointers to other nodes and variables. A variable is an array of data of a given type and length. A more detailed description is given below.

3.1 Nodes

A node represents a collection of related information. Associated with a given node are node header information, variables and subnodes. The top node in the file is called the root node and is created automatically when the file is created. All other nodes must be explicitly created.

All nodes except the root node must be given a name by the creating application. Names may be up to 8 characters long for versions 1 and 2 and up to 32 characters long for version 3. Names may contain any character, however the use of non-displayable characters may prevent portability. Upper case and lower case characters are considered different (i.e., `Zone 1` is not equal to `ZONE 1`). Examples of names are `ZONE 1` and `BNDRY 1`.

By default, a node may have up to 64 subnodes and 64 variables to which it directly associates. Subnodes of subnodes or variables of subnodes do not count toward this limit. The maximum number of subnodes and variables can be increased at the time the file is created subject to a library imposed maximum. For version 3 files there is no node limit. The limit now represents only your desired program limit and can be changed at any time.

3.2 Node Header Information

Each node has associated with it node header information. The node header information is provided for the application to store a small amount of information about the node. The contents of the node header are application dependent and data within the node header can be reliably used by different applications only if they agree on its organization.

By default, up to 64 integer values, 64 single precision floating point values, and two 80 character strings can be stored in a node header. The number of each value can be increased at file creation time subject to a maximum imposed by the library.

3.3 Variables

Variables contain the data to be written or read by the application. A variable may be of type `INTEGER`, `REAL`, or `DOUBLE PRECISION` and is associated with a specific node. For version 3 files a new `CHARACTER` variable has been added. A variable is known only within the node with which it is associated, i.e., variable `X` associated with node `A` has no relation to variable `X` in node `B`. It can be of a different type or length.

Like nodes, variables have names and are subject to the same restrictions. Examples of names are `x`, `rho*u` and `T`.

4 Reference and Scaling Data

Reference and scaling data are used in common files to provide a mechanism for applications to exchange data even if they use different sets of units for dimensional data. The understanding and use of this concept is key when creating applications that read and/or write common files.

Two pairs of numbers are required. The first pair is the reference data, and specifies how the data is non-dimensionalized. Both a reference non-dimensionalization factor and a reference offset value are required. The offset value allows data to be stored in the form of a difference from some value, such as $p - p_\infty$. The second pair of numbers is the scaling data, and specifies how to convert the dimensional data to SI units. Again, both a scaling factor and a scaling offset value are required. The basic SI units are meters, kilograms, seconds, and degrees Kelvin, for length, mass, time, and temperature, respectively. The derived SI units newtons and joules are used for force and energy.

Thus, for any variable V, if the value stored in the file is V_{file} a dimensional value may be computed from

$$V_{dim} = V_{file}\, RF_V + RO_V$$

where RF_V and RO_V are the reference factor and offset value for the variable V. Then, the value in SI units may be computed from

$$V_{SI} = V_{dim}\, SF_V + SO_V$$

where SF_V and SO_V are the scaling factor and offset value for V.

Note that when dimensional data is stored in the file, the value of the reference factor RF will be 1.0. Note also that when reading a common file, the units for a variable may be determined by the value of its scaling factor SF. For example the units for the x coordinate are meters if its scaling factor is 1.0, inches if its scaling factor is 0.0254, etc.

Reference and scaling data are attached to a specific variable name. At the current time, the data are applied on a file-wide basis. I.e., like-named variables under different nodes use the same reference and scaling data. (At some point in the future reference and scaling data may be applicable on a node-by-node basis.)

For current common files (i.e., Version 3) the reference and scaling data are stored in a subnode of the root node named `RefScl00000001_CFF`. There are two arrays there, `VSCLST` and `REFARR`. `VSCLST` is a character array containing the names of the variables in the file, with room for 512 names and 32 characters per name. `REFARR` is an array of 512 single-precision real values containing the reference and scaling data for the variables in `VSCLST`. For each variable, the values in `REFARR` are ordered as *SO*, *SF*, *RO*, *RF*.

For example, if temperature is stored in the file non-dimensionally as $(T - T_\infty)/T_\infty$, where all values are in degrees Rankine, the reference and scaling data would be:

$SO = 0.0$
$SF = 5.0/9.0$
$RO = T_\infty$
$RF = T_\infty$

5 Subroutine Descriptions

This section describes in detail the subroutine calls that define the common file library. All of the subroutine calls (with the exception of CFERMS and CFINIT) have the following format using the Fortran interface:

```
CALL CFxxxx (STATUS, STATE, ...)
```

where STATUS is an integer value returned by the routine to indicate success or failure of the request and STATE is an integer variable corresponding to the file state to which the request is to be applied. A file state represents a particular file and node to which a common file request is applied. The application must initialize the STATE variable to zero before the first call to CFSTFP or CFRECL for a specific file. Thereafter the library routines maintain the STATE variable. Note that the application must not modify the STATE variable.

By default, all of the common file routines abort the program if an error is detected except CFINIT, CFERMS, CFVINF, and CFNINF. The program may prevent this action by calling CFSTFP to set the ABORT flag to zero. This will cause the library routines to return a non-zero status value if an error is detected.

The common file library now supports a C interface. The C interface has the same arguments, with the same meanings, as the Fortran interface, and the C routine names are the same as the Fortran routine names with the leading "CF" replaced by "CF_C". The type definitions and defines for the C interface are in the *cfbind_c_2_f.h* include file.

The following sections show the calling statement for each subroutine, with the Fortran version listed first, followed by the C version in italics. The arguments are then defined, and classified as *Input*, *Output*, or *Temporary*. In a few cases, an argument has two definitions, one for input and another for output. The argument names include the data type (i.e., integer, character, etc.), and again the Fortran name is listed first, followed by the C name in italics.

5.1 Initialization Functions

`CFINIT` — Initialize the Common File Library

```
CALL CFINIT ()
void CF_Cinit ()
```

`CFINIT` initializes various internal tables used by the common file library routines. It must be called at least once in a program and must be called before any other common file library routine. Multiple calls to this subroutine will have no effect on the common file tables or operation.

5.2 File Oriented Functions

CFSTFP — Set a File Parameter

```
CALL CFSTFP (STATUS, STATE, PARNAM, PARVAL)
Fint CF_Cstfp (status, state, parnam, parval)
```

INTEGER STATUS *Fint *status*	*Output*	Return status; for C it is also the function return.
INTEGER STATE *Fint *state*	*Input*	The root state of the file or the state variable initialized to zero if no previous calls to CFSTFP, CFGTFP, or CFRECL have been made to this file.
	Output	The root state if STATE was zero.
CHARACTER*(*) PARNAM *Fchar *parnam*	*Input*	The name of the parameter to be set.
INTEGER PARVAL *Fint parval*	*Input*	The value to be assigned to the parameter specified in PARNAM.

CFSTFP is used to set the file parameters for use by a subsequent CFOPEN request. The following parameters can be set:

UNIT	No default, *must* be set except for version 3
CIO	Use C I/O interface, limited support, not used for version 3
ABORT	1 (abort if error occurs)
FILE_CPU	File CPU type
FILE_OS	File operating system type
RECORD_LENGTH	Depends on host CPU and operating system
MAX_NODES	Maximum number of nodes per subnode
MAX_VARIABLES	Maximum number of variables per node
MAX_INTEGERS	Number of INTEGER elements in a node header
MAX_REALS	Number of REAL elements in a node header
MAX_CHARACTERS	Number of CHARACTER*80 elements in a node header
FILE_VERSION	Version of the file to create must equal 2 or 3 (default)

The values of RECORD_LENGTH, MAX_NODES, MAX_VARIABLES, MAX_INTEGERS, MAX_REALS, and MAX_CHARACTERS must be a multiple of two to allow for portability between machines and are subject to limitations described in Section 7, "Default Library Values and Limits," on p. 53. For version 3 files the unit number is not used since the open is done in C.

If a file is opened and a specific parameter has not been set by CFSTFP then a default value will be assigned for the parameter. The default values are also defined in Section 7.

Only the ABORT, FILE_CPU, and FILE_OS parameters may be set after the file is opened. For version 3 files the MAX_NODES and MAX_VARIABLES parameters may be changed after open to redefine the maximum that is allowed to be created in the file.

CFGTFP — Get a File Parameter

```
CALL CFGTFP (STATUS, STATE, PARNAM, PARVAL)
Fint CF_Cgtfp (status, state, parnam, parval)
```

INTEGER STATUS *Fint *status*	*Output*	Return status; for C it is also the function return.
INTEGER STATE *Fint *state*	*Input*	The root state of the file or the state variable initialized to zero if no previous calls to CFSTFP, CFGTFP or CFRECL have been made to this file.
	Output	The root state if STATE was zero.
CHARACTER*(*) PARNAM *Fchar *parnam*	*Input*	Name of the parameter whose value is to be returned.
INTEGER PARVAL *Fint *parval*	*Output*	Value of the requested parameter.

CFGTFP retrieves the value of a file parameter. If called prior to opening a file then the value returned will be either the default value for the parameter or the value set by CFSTFP. If called after the file is opened then the value will be that contained in the file header.

The parameter name may be any one of the following names:

```
UNIT
OPEN
HOST_CPU
HOST_OS
ABORT
FILE_CPU
FILE_OS
RECORD_LENGTH
MAX_NODES
MAX_VARIABLES
MAX_INTEGERS
MAX_REALS
MAX_CHARACTERS
LAST_RECORD
VERSION
```

Note that FILE_CPU, FILE_OS and LAST_RECORD are not known until the file is opened.

CFRECL — Set Record Length to be Used Before Calling CFOPEN

```
CALL CFRECL (STATUS, STATE, MAXPTS, PRECSN)
Fint CF_Crecl (status, state, maxpts, precsn)
```

INTEGER STATUS *Fint *status*	*Output*	Return status; for C it is also the function return.
INTEGER STATE *Fint *state*	*Input*	The root state of the file or the state variable initialized to zero if no previous calls to CFSTFP, CFGTFP or CFRECL have been made to this file.
	Output	The root state if STATE was zero.
INTEGER MAXPTS *Fint maxpts*	*Input*	Maximum variable size.
INTEGER PRECSN *Fint precsn*	*Input*	Precision of the proposed record. 1 = Real or Integer. 2 = Double precision.

CFRECL sets the record length to be used when creating a common file with a subsequent CFOPEN request. The routine uses MAXPTS and PRECSN to calculate a record length acceptable to the operating system. This works fine for simple one size variable files but not very well for files containing variables of many different sizes. Thus the CFOREC routine (see p. 46) should be used to calculate record lengths for an arbitrary sized file. For version 3 files there is no record length although the record length parameter stored in the file is still used to determine the size of the conversion buffers used in the CFRV*xx* subroutines.

For example, assume you are going to be writing single precision records dimensioned IDIM × JDIM × KDIM. The following call will set the record length:

```
RSTATE = 0
CALL CFRECL (STATUS, RSTATE, IDIM*JDIM*KDIM, 1)
```

`CFOPEN` — Open a Common File

```
CALL CFOPEN (STATUS, STATE, FILNAM, MODE)
Fint CF_Copen (status, state, filnam, mode)
```

`INTEGER STATUS` *`Fint *status`*	*Output*	Return status; for C it is also the function return.
`INTEGER STATE` *`Fint *state`*	*Input*	The state from a previous `CFSTFP` or `CFRECL` call for the file.
	Output	The state for the root node of the file.
`CHARACTER*(*) FILNAM` *`Fchar *filnam`*	*Input*	The name of the file to be opened. Any extension must be included (*.cgd*, *.cfl*, etc.). `FILNAM` is ignored if `MODE` is specified as `SCRATCH`. If `FILNAM` is all blanks, a file name of `FOR0`***`uu`*** (where ***`uu`*** is the unit number set by `CFSTFP`) will be used.
`CHARACTER*(*) MODE` *`Fchar *mode`*	*Input*	How the file is to be opened. `MODE` must be `OLD`, `NEW`, or `SCRATCH`.

`CFOPEN` allocates internal tables for the common file library and opens the file for processing.

A `CFOPEN` call must be preceded, at minimum, by a call to `CFSTFP` to set the unit number. For version 3 files the unit number is not used, thus the `CFSTFP` call is unnecessary in the following example of a minimal opening sequence for a common file:

```
INTEGER STATE, STATUS
STATE = 0
CALL CFINIT ()
CALL CFSTFP (STATUS, STATE, 'UNIT', 11)
CALL CFOPEN (STATUS, STATE, 'TEST.CGD', 'NEW')
```

When `CFOPEN` is called with a mode of `NEW` or `SCRATCH`, the attributes of the file are set from either default values or from the values set by calls to `CFSTFP`. The default values are listed in Section 7, "Default Library Values and Limits," on p. 53.

When `CFOPEN` is called with a mode of `OLD`, the default values or the values specified by `CFSTFP` are used to verify that the calling program can accomodate the file being opened and are subsequently overwritten by the actual values from the file. Thus if the program let everything default and tried to open a file with `MAX_INTEGERS` set to 128, the request to open the file would be rejected.

The only routines that can be called prior to `CFOPEN` for a given file are `CFSTFP`, `CFRECL`, `CFOREC`, `CFUNIT`, and `CFERMS`.

CFCLOS — Close a Common File

```
CALL CFCLOS (STATUS, STATE)
Fint CF_Cclos (status, state)
```

INTEGER STATUS *Fint *status*	*Output*	Return status; for C it is also the function return.
INTEGER STATE *Fint *state*	*Input*	The root state (from CFOPEN) of the file to be closed.
	Output	Zero.

CFCLOS closes a common file. Any internal table space within the common file library is released and made available for use for other files.

5.3 Node Oriented Functions

CFCNOD — Create a Node

```
CALL CFCNOD (STATUS, STATE, NODNAM, NSTATE)
Fint CF_Ccnod (status, state, nodnam, nstate)
```

INTEGER STATUS *Fint *status*	*Output*	Return status; for C it is also the function return.
INTEGER STATE *Fint *state*	*Input*	The state of the node that will be the parent node of the subnode that is being created.
CHARACTER*(*) NODNAM *Fchar *nodnam*	*Input*	Name of the subnode to be created. It will be truncated or padded with blanks to 8 (for versions 1 and 2) or 32 (for version 3) characters as required. The name is case-sensitive (i.e., x is not equal to X).
INTEGER NSTATE *Fint *nstate*	*Output*	The state of the newly created subnode.

CFCNOD is used to create a subnode under another node. The following code fragment creates a subnode under the root node using the name stored in ZNAME (RSTATE is the root state returned by CFOPEN):

```
CHARACTER*32 ZNAME
INTEGER ZONE,ZSTATE,RSTATE
ZNAME = subnode_name
CALL CFCNOD (STATUS, RSTATE, ZNAME, ZSTATE)
```

Once the request has been successfully completed, the program may read and write the node header information and variables of the subnode by using the the new state (ZSTATE) as the state for subsequent calls to CFR/WNOD and CFR/WV*x*NOD.

The integer and real elements in the node header will be initialized to zero and the character elements will be initialized to blank.

CFCLNK — Create a Node Link

```
CALL CFCLNK (STATUS, STATE, NODNAM, FILNAM, LNKNAM, NSTATE)
Fint CF_Clnk (status, state, nodnam, filnam, lknam, nstate)
```

INTEGER STATUS *Fint *status*	*Output*	Return status; for C it is also the function return.
INTEGER STATE *Fint *state*	*Input*	The state of the node that will be the parent node of the subnode that is being created.
CHARACTER*(*) NODNAM *Fchar *nodnam*	*Input*	Name of the subnode to be created. It will be truncated or padded with blanks to 8 (for versions 1 and 2) or 32 (for version 3) characters as required. The name is case-sensitive (i.e., x is not equal to X).
CHARACTER*(*) FILNAM *Fchar *filnam*	*Input*	The name of the file to link to; if blank, the link will be to a node in the current file.
CHARACTER*(*) LNKNAM *Fchar *lknam*	*Input*	The name of the node to link to.
INTEGER NSTATE *Fint *nstate*	*Output*	The state of the newly created subnode.

CFCLNK is used to create a subnode which points to another node. The node may be in the current file or another file. The FILNAM can contain a path, but it is not recommended since the user does not know what the path is, and so does not know where the file should be located so the library can open it. The following code fragment creates a link under the root node called ZNAME which points to node "Zone 1" in file *ZONE1.cgd* (RSTATE is the root state returned by CFOPEN):

```
CHARACTER*32 ZNAME,LNKNAM,FILNAM
INTEGER ZONE,ZSTATE,RSTATE
ZNAME = 'Node 1'
FILNAM = 'ZONE1.cgd'
LNKNAM = '/ZONE   1'
CALL CFCLNK (STATUS, RSTATE, ZNAME, FILNAM, LNKNAM, ZSTATE)
```

`CFNDEL` — Delete a Node

```
CALL CFNDEL (STATUS, STATE, NODNAM)
Fint CF_Cndel (status, state, nodnam)
```

`INTEGER STATUS` *`Fint *status`*	*Output*	Return status; for C it is also the function return.
`INTEGER STATE` *`Fint *state`*	*Input*	The state of the node that is the parent node of the subnode that is being deleted.
`CHARACTER*(*) NODNAM` *`Fchar *nodnam`*	*Input*	Name of the subnode to be deleted. It will be truncated or padded with blanks to 8 (for versions 1 and 2) or 32 (for version 3) characters as required. The name is case-sensitive (i.e., `x` is not equal to `X`).

`CFNDEL` is used to delete a subnode under another node. The following code fragment deletes a subnode under the root node using the name stored in `ZNAME` (`RSTATE` is the root state returned by `CFOPEN`):

```
CHARACTER*32 ZNAME
INTEGER ZONE,RSTATE
ZNAME = subnode_name
CALL CFNDEL (STATUS, RSTATE, ZNAME)
```

Once the request has been successfully completed, all data for the node and all subnodes under that node will be inaccessible. For version 1 and 2 files the data is not removed and the space becomes dead space in the file. For version 3 files the ADF core puts the space into a free chunk table to be reallocated to a new node or variable. The actual size of the file does not change.

`CFNMOV` — Move a Node

```
CALL CFNMOV (STATUS, STATE, NODNAM, TSTATE)
Fint CF_Cndel (status, state, nodnam, tstate)
```

`INTEGER STATUS` *`Fint *status`*	*Output*	Return status; for C it is also the function return.
`INTEGER STATE` *`Fint *state`*	*Input*	The state of the node that is the parent node of the subnode that is being moved.
`CHARACTER*(*) NODNAM` *`Fchar *nodnam`*	*Input*	Name of the subnode to be moved. It will be truncated or padded with blanks to 8 (for versions 1 and 2) or 32 (for version 3) characters as required. The name is case-sensitive (i.e., `x` is not equal to `X`).
`INTEGER TSTATE` *`Fint *tstate`*	*Input*	The state of the new parent node for `NODNAM`.

`CFNMOV` is used to move a subnode from one node to another in the same file. All subnodes of the moved node move with it. No data in the file is actually moved; only the node pointers are moved.

`CFNREN` — Rename a Node

```
CALL CFNREN (STATUS, STATE, NODNAM, NEWNAM)
Fint CF_Cnren (status, state, nodnam, newnam)
```

`INTEGER STATUS` *`Fint *status`*	*Output*	Return status; for C it is also the function return.
`INTEGER STATE` *`Fint *state`*	*Input*	The state of the node that is the parent node of the subnode that is being renamed.
`CHARACTER*(*) NODNAM` *`Fchar *nodnam`*	*Input*	Name of the subnode to be renamed. It will be truncated or padded with blanks to 8 (for versions 1 and 2) or 32 (for version 3) characters as required. The name is case-sensitive (i.e., `x` is not equal to `X`).
`CHARACTER*(*) NEWNAM` *`Fchar *newnam`*	*Input*	The new name of the node. It will be truncated or padded with blanks to 8 (for versions 1 and 2) or 32 (for version 3) characters as required. The name is case-sensitive (i.e., `x` is not equal to `X`).

`CFNREN` is used to rename a subnode of a node. The new name must be unique under the parent node.

CFSNOD — Establish Access to a Node

```
CALL CFSNOD (STATUS, STATE, NODNAM, NSTATE)
Fint CF_Csnod (status, state, nodnam, nstate)
```

INTEGER STATUS *Fint *status*	*Output*	Return status; for C it is also the function return.
INTEGER STATE *Fint *state*	*Input*	State of the node containing the specified subnode.
CHARACTER*(*) NODNAM *Fchar *nodnam*	*Input*	Name of the subnode to be accessed. It will be truncated or padded with blanks to 8 (for versions 1 and 2) or 32 (for version 3) characters as required. The name is case-sensitive (i.e., x is not equal to X).
INTEGER NSTATE *Fint *nstate*	*Output*	The node state of the requested subnode.

CFSNOD is used access a subnode under another node. The following code fragment accesses a subnode under the root node using the name stored in ZNAME (RSTATE is the root state returned by CFOPEN):

```
CHARACTER*32 ZNAME
INTEGER ZONE,ZSTATE,RSTATE
ZNAME = subnode_name
CALL CFSNOD (STATUS, RSTATE, ZNAME, ZSTATE)
```

Once the request has been successfully completed, the program may read and write the node header information and variables of the subnode by using the the new state (ZSTATE) as the state for subsequent calls to CFR/WNOD and CFR/WV*x*NOD.

`CFRNOD` — Read Node Header Information

```
CALL CFRNOD (STATUS, STATE, IPAR, FPAR, CPAR)
Fint CF_Crnod (status, state, ipar, fpar, cpar)
```

`INTEGER STATUS` *`Fint *status`*	*Output*	Return status; for C it is also the function return.
`INTEGER STATE` *`Fint *state`*	*Input*	Node state of the node whose header information is to be read.
`INTEGER IPAR(*)` *`Fint *ipar`*	*Output*	The integer elements from the node header. The size of the array must be at least `MAX_INTEGERS` long (default 64, see CFSTFP).
`REAL FPAR(*)` *`Freal *fpar`*	*Output*	The real elements from the node header. The size of the array must be at least `MAX_REALS` long (default 64, see CFSTFP).
`CHARACTER*80 CPAR(*)` *`Fchar **cpar`*	*Output*	The character elements from the node header. The size of the array must be at least `MAX_CHARACTERS` long (default 2, see CFSTFP).

`CFRNOD` reads the application dependent node header information for the specified node.

CFWNOD — Write Node Header Information

```
CALL CFWNOD (STATUS, STATE, IPAR, FPAR, CPAR)
Fint CF_Cwnod (status, state, ipar, fpar, cpar)
```

INTEGER STATUS *Fint *status*	*Output*	Return status; for C it is also the function return.
INTEGER STATE *Fint *state*	*Input*	Node state of the node whose header information is to be written.
INTEGER IPAR(*) *Fint *ipar*	*Input*	The integer elements from the node header. The size of the array must be at least MAX_INTEGERS long (default 64, see CFSTFP).
REAL FPAR(*) *Freal *fpar*	*Input*	The real elements from the node header. The size of the array must be at least MAX_REALS long (default 64, see CFSTFP).
CHARACTER*80 CPAR(*) *Fchar **cpar*	*Input*	The character elements from the node header. The size of the array must be at least MAX_CHARACTERS long (default 2, see CFSTFP).

CFWNOD writes the application dependent node header information for the specified node.

`CFNINF` — Get Information About a Node

```
CALL CFNINF (STATUS, STATE, NODNAM, NNODES, NVARS)
Fint CF_Cninf (status, state, nodnam, nnodes, nvars)
```

`INTEGER STATUS` *`Fint *status`*	*Output*	Return status; for C it is also the function return.
`INTEGER STATE` *`Fint *state`*	*Input*	Node state of the node.
`CHARACTER*(*) NODNAM` *`Fchar *nodnam`*	*Input*	Name of the node. It will be truncated or padded with blanks to 8 (for versions 1 and 2) or 32 (for version 3) characters as required. The name is case-sensitive (i.e., `x` is not equal to `X`).
`INTEGER NNODES` *`Fint *nnodes`*	*Output*	Number of subnodes under the node.
`INTEGER NVARS` *`Fint *nvars`*	*Output*	Number of variables under the node.

`CFNINF` returns information about a node. Note that if the requested node is not present, `CFNINF` will not abort even if the `ABORT` flag is set.

5.4 Data Read and Write Functions

CFRVI — Read Integer Variable

```
CALL CFRVI (STATUS, STATE, VARNAM, VARLEN, INCR, VARDAT)
Fint CF_Crvi (status, state, varnam, varlen, incr, vardat)
```

INTEGER STATUS *Fint *status*	*Output*	Return status; for C it is also the function return.
INTEGER STATE *Fint *state*	*Input*	Node state of the node which contains the requested variable.
CHARACTER*(*) VARNAM *Fchar *varnam*	*Input*	Name of the variable to be read. It will be truncated or padded with blanks to 8 (for versions 1 and 2) or 32 (for version 3) characters as required. The name is case-sensitive (i.e., x is not equal to X).
INTEGER VARLEN *Fint *varlen*	*Output*	Number of elements read.
INTEGER INCR *Fint incr*	*Input*	Increment between elements to be applied when moving data to the output array.
INTEGER VARDAT(*) *Fint *vardat*	*Output*	The data read from the file.

CFRVI reads an integer variable located under the specified node.

`CFRVR` — Read Real Variable

```
CALL CFRVR (STATUS, STATE, VARNAM, VARLEN, INCR, VARDAT)
Fint CF_Crvr (status, state, varnam, varlen, incr, vardat)
```

`INTEGER STATUS` *`Fint *status`*	*Output*	Return status; for C it is also the function return.
`INTEGER STATE` *`Fint *state`*	*Input*	Node state of the node which contains the requested variable.
`CHARACTER*(*) VARNAM` *`Fchar *varnam`*	*Input*	Name of the variable to be read. It will be truncated or padded with blanks to 8 (for versions 1 and 2) or 32 (for version 3) characters as required. The name is case-sensitive (i.e., `x` is not equal to `X`).
`INTEGER VARLEN` *`Fint *varlen`*	*Output*	Number of elements read.
`INTEGER INCR` *`Fint incr`*	*Input*	Increment between elements to be applied when moving data to the output array.
`REAL VARDAT(*)` *`Freal *vardat`*	*Output*	The data read from the file.

`CFRVR` reads a real variable located under the specified node.

CFRVD — Read Double Precision Variable

```
CALL CFRVD (STATUS, STATE, VARNAM, VARLEN, INCR, VARDAT)
Fint CF_Crvd (status, state, varnam, varlen, incr, vardat)
```

INTEGER STATUS *Fint *status*	*Output*	Return status; for C it is also the function return.
INTEGER STATE *Fint *state*	*Input*	Node state of the node which contains the requested variable.
CHARACTER*(*) VARNAM *Fchar *varnam*	*Input*	Name of the variable to be read. It will be truncated or padded with blanks to 8 (for versions 1 and 2) or 32 (for version 3) characters as required. The name is case-sensitive (i.e., x is not equal to X).
INTEGER VARLEN *Fint *varlen*	*Output*	Number of elements read.
INTEGER INCR *Fint incr*	*Input*	Increment between elements to be applied when moving data to the output array.
DOUBLE PRECISION VARDAT(*) *Fdouble *vardat*	*Output*	The data read from the file.

CFRVD reads a double precision precision variable located under the specified node.

CFRVRX — Read Real Variable with Conversion

```
CALL CFRVRX (STATUS, STATE, VARNAM, VARLEN, INCR, VARDAT, TMPBUF)
Fint CF_Crvrx (status, state, varnam, varlen, incr, vardat)
```

INTEGER STATUS *Fint *status*	*Output*	Return status; for C it is also the function return.
INTEGER STATE *Fint *state*	*Input*	Node state of the node containing the requested variable.
CHARACTER*(*) VARNAM *Fchar *varnam*	*Input*	Name of the variable to be read. It will be truncated or padded with blanks to 8 (for versions 1 and 2) or 32 (for version 3) characters as required. The name is case-sensitive (i.e., x is not equal to X).
INTEGER VARLEN *Fint *varlen*	*Output*	Number of elements read.
INTEGER INCR *Fint incr*	*Input*	Increment between elements to be applied when moving data to the output array.
REAL VARDAT(*) *Freal *vardat*	*Output*	The data read from the file.
DOUBLE PRECISION TMPBUF(*)	*Temporary*	A temporary buffer capable of containing a record length of real numbers. The record length can be obtained from CFGTFP.

CFRVRX reads either a real or double precision value located under the specified node into a single precision array. If the requested variable is a single precision variable then the call is equivalent to a call to CFRVR. If the requested variable is double precision then the data is read in and converted to single precision in the output array.

CFRVDX — Read Double Precision Variable with Conversion

```
CALL CFRVDX (STATUS, STATE, VARNAM, VARLEN, INCR, VARDAT, TMPBUF)
Fint CF_Crvdx (status, state, varnam, varlen, incr, vardat)
```

INTEGER STATUS *Fint *status*	*Output*	Return status; for C it is also the function return.
INTEGER STATE *Fint *state*	*Input*	Node state of the node containing the requested variable.
CHARACTER*(*) VARNAM *Fchar *varnam*	*Input*	Name of the variable to be read. It will be truncated or padded with blanks to 8 (for versions 1 and 2) or 32 (for version 3) characters as required. The name is case-sensitive (i.e., x is not equal to X).
INTEGER VARLEN *Fint *varlen*	*Output*	Number of elements read.
INTEGER INCR *Fint incr*	*Input*	Increment between elements to be applied when moving data to the output array.
DOUBLE PRECISION VARDAT(*) *Fdouble *vardat*	*Output*	The data read from the file.
REAL TMPBUF(*)	*Temporary*	A temporary buffer capable of containing a record length of real numbers. The record length can be obtained from CFGTFP.

CFRVDX reads either a real or double precision value located under the specified node into a double precision array. If the requested variable is a double precision variable then the call is equivalent to a call to CFRVD. If the requested variable is single precision then the data is read in and converted to double precision in the output array.

`CFRVC` — Read Character Variable

```
CALL CFRVC (STATUS, STATE, VARNAM, VARLEN, CHRSIZ, INCR, VARDAT)
Fint CF_Crvc (status, state, varnam, varlen, chrsiz, incr, vardat)
```

`INTEGER STATUS` *`Fint *status`*	*Output*	Return status; for C it is also the function return.
`INTEGER STATE` *`Fint *state`*	*Input*	Node state of the node containing the requested variable.
`CHARACTER*(*) VARNAM` *`Fchar *varnam`*	*Input*	Name of the variable to be read. It will be truncated or padded with blanks to 32 characters as required. The name is case-sensitive (i.e., `x` is not equal to `X`).
`INTEGER VARLEN` *`Fint *varlen`*	*Output*	Number of elements read.
`INTEGER CHRSIZ` *`Fint *chrsiz`*	*Input*	The size of the character elements. `CHRSIZ` is used to increment through the data elements for Fortran character arrays.
`INTEGER INCR` *`Fint incr`*	*Input*	Increment between elements to be applied when moving data to the output array.
`CHARACTER*(*) VARDAT(*)` *`Fchar *vardat`*	*Output*	The data read from the file.

`CFRVC` reads a character variable located under the specified node. This function is only available in version 3 or greater.

`CFWVI` — Write Integer Variable

```
CALL CFWVI (STATUS, STATE, VARNAM, VARLEN, INCR, VARDAT)
Fint CF_Cwvi (status, state, varnam, varlen, incr, vardat)
```

`INTEGER STATUS` *`Fint *status`*	*Output*	Return status; for C it is also the function return.
`INTEGER STATE` *`Fint *state`*	*Input*	Node state of the node containing the requested variable.
`CHARACTER*(*) VARNAM` *`Fchar *varnam`*	*Input*	Name of the variable to be written. It will be truncated or padded with blanks to 8 (for versions 1 and 2) or 32 (for version 3) characters as required. The name is case-sensitive (i.e., `x` is not equal to `X`).
`INTEGER VARLEN` *`Fint varlen`*	*Input*	Number of elements to write.
`INTEGER INCR` *`Fint incr`*	*Input*	Increment between elements to be applied when writing the data to the file.
`INTEGER VARDAT(*)` *`Fint *vardat`*	*Input*	The data to be written.

`CFWVI` writes an integer variable under the specified node. The variable will be created if it does not already exist. If the variable already exists, the new contents will replace the old contents. The old length and the new length do not have to be equal.

`CFWVR` — Write Real Variable

```
CALL CFWVR (STATUS, STATE, VARNAM, VARLEN, INCR, VARDAT)
Fint CF_Cwvr (status, state, varnam, varlen, incr, vardat)
```

`INTEGER STATUS` *`Fint *status`*	*Output*	Return status; for C it is also the function return.
`INTEGER STATE` *`Fint *state`*	*Input*	Node state of the node containing the requested variable.
`CHARACTER*(*) VARNAM` *`Fchar *varnam`*	*Input*	Name of the variable to be written. It will be truncated or padded with blanks to 8 (for versions 1 and 2) or 32 (for version 3) characters as required. The name is case-sensitive (i.e., `x` is not equal to `X`).
`INTEGER VARLEN` *`Fint varlen`*	*Input*	Number of elements to write.
`INTEGER INCR` *`Fint incr`*	*Input*	Increment between elements to be applied when writing the data to the file.
`REAL VARDAT(*)` *`Freal *vardat`*	*Input*	The data to be written.

`CFWVR` writes a real variable under the specified node. The variable will be created if it does not already exist. If the variable already exists, the new contents will replace the old contents. The old length and new length do not have to be equal.

If the variable already exists and is double precision, the data will be written as double precision. The input data will be unaffected.

CFWVD — Write Double Precision Variable

```
CALL CFWVD (STATUS, STATE, VARNAM, VARLEN, INCR, VARDAT)
Fint CF_Cwvd (status, state, varnam, varlen, incr, vardat)
```

INTEGER STATUS *Fint *status*	*Output*	Return status; for C it is also the function return.
INTEGER STATE *Fint *state*	*Input*	Node state of the node containing the requested variable.
CHARACTER*(*) VARNAM *Fchar *varnam*	*Input*	Name of the variable to be written. It will be truncated or padded with blanks to 8 (for versions 1 and 2) or 32 (for version 3) characters as required. The name is case-sensitive (i.e., x is not equal to X).
INTEGER VARLEN *Fint varlen*	*Input*	Number of elements to write.
INTEGER INCR *Fint incr*	*Input*	Increment between elements to be applied when writing the data to the file.
DOUBLE PRECISION VARDAT(*) *Fdouble *vardat*	*Input*	The data to be written.

CFWVD writes a double precision variable under the specified node. The variable will be created if it does not already exist. If the variable already exists, the new contents will replace the old contents. The old length and new length do not have to be equal.

If the variable exists and is single precision, the data will be written as single precision. The input data will be unaffected.

`CFWVC` — Write Character Variable

```
CALL CFWVC (STATUS, STATE, VARNAM, VARLEN, CHRSIZ, INCR, VARDAT)
Fint CF_Cwvc (status, state, varnam, varlen, chrsiz, incr, vardat)
```

`INTEGER STATUS` *`Fint *status`*	*Output*	Return status; for C it is also the function return.
`INTEGER STATE` *`Fint *state`*	*Input*	Node state of the node containing the requested variable.
`CHARACTER*(*) VARNAM` *`Fchar *varnam`*	*Input*	Name of the variable to be written. It will be truncated or padded with blanks to 32 characters as required. The name is case-sensitive (i.e., `x` is not equal to `X`).
`INTEGER VARLEN` *`Fint varlen`*	*Output*	Number of elements of size CHRSIZ to write.
`INTEGER CHRSIZ` *`Fint *chrsiz`*	*Input*	The size of the character elements. The total length written will be `VARLEN*CHRSIZ`. `CHRSIZ` is also used to increment through the data elements for Fortran character arrays.
`INTEGER INCR` *`Fint incr`*	*Input*	Increment between elements to be applied when moving data from the output array.
`CHARACTER*(*) VARDAT(*)` *`Fchar *vardat`*	*Input*	The data written to the file.

`CFWVC` writes a character variable located under the specified node. This function is only available in version 3 or greater.

CFVDEL — Delete a Variable

```
CALL CFVDEL (STATUS, STATE, VARNAM)
Fint CF_Cvdel (status, state, varnam)
```

INTEGER STATUS *Fint *status*	*Output*	Return status; for C it is also the function return.
INTEGER STATE *Fint *state*	*Input*	The state of the node that contains the variable that is being deleted.
CHARACTER*(*) VARNAM *Fchar *varnam*	*Input*	Name of the variable to be deleted. It will be truncated or padded with blanks to 8 (for versions 1 and 2) or 32 (for version 3) characters as required. The name is case-sensitive (i.e., x is not equal to X).

CFVDEL is used to delete a variable under a node. Once the request has been successfully completed, all data for the variable will be inaccessible. For version 1 and 2 files the data is not removed and the space becomes dead space in the file. For version 3 files the ADF core puts the space into a free chunk table to be reallocated to a new node or variable. The actual size of the file does not change.

5.5 Miscellaneous Functions

`CFERMS` — Retrieve Error Message Text for Specified Error Number

```
CALL CFERMS (ERRNUM, ERRTXT)
Fint CF_Cerms (errnum, errtxt)
```

`INTEGER ERRNUM` *Input* Error number for which text is being requested.
`Fint errnum`

`CHARACTER*(*) ERRTXT` *Output* Text for specified error number.
`char *errtxt`

`CFERMS` returns a textual error message corresponding to an error number returned by a call to a common file library routine. See Section 6, "Status Codes," starting on p. 51 for the error numbers and their associated text. For version 3 files the ADF core errors are returned as a negative number to distinguish them from the common file error numbers. The ADF error message routine is automatically called to return the proper error string.

CFRREF — Read Reference and Scaling Data

```
CALL CFRREF (STATUS, STATE, VARNAM, UCOFST, UCSCAL, NDOFST, NDSCAL)
Fint CF_Crref (status, state, varnam, ucofst, ucscal, ndofst, ndscal)
```

INTEGER STATUS *Fint *status*	*Output*	Return status; for C it is also the function return.
INTEGER STATE *Fint *state*	*Input*	The root state of the file.
CHARACTER*(*) VARNAM *Fchar *varnam*	*Input*	Name of the variable to obtain reference data. It will be truncated or padded with blanks to 8 (for versions 1 and 2) or 32 (for version 3) characters as required. The name is case-sensitive (i.e., x is not equal to X).
REAL UCOFST *Freal *ucofst*	*Output*	Unit conversion offset.
REAL UCSCAL *Freal *ucscal*	*Output*	Unit conversion scale factor.
REAL NDOFST *Freal *ndofst*	*Output*	Non-dimensionalization offset.
REAL NDSCAL *Freal *ndscal*	*Output*	Non-dimensionalization scale factor.

CFRREF reads the reference and scaling data for the specified variable. If no reference and scaling data has been written for the specified variable then the following values are returned:

```
UCOFST = 0.0
UCSCAL = 1.0
NDOFST = 0.0
NDSCAL = 1.0
```

See Section 4 for a description of the function of the reference and scaling data.

`CFWREF` — Write Reference and Scaling Data

```
CALL CFWREF (STATUS, STATE, VARNAM, UCOFST, UCSCAL, NDOFST, NDSCAL)
Fint CF_Cwref (status, state, varnam, ucofst, ucscal, ndofst, ndscal)
```

`INTEGER STATUS` *`Fint *status`*	*Output*	Return status; for C it is also the function return.
`INTEGER STATE` *`Fint *state`*	*Input*	The root state of the file.
`CHARACTER*(*) VARNAM` *`Fchar *varnam`*	*Input*	Name of the variable to write reference data. It will be truncated or padded with blanks to 8 (for versions 1 and 2) or 32 (for version 3) characters as required. The name is case-sensitive (i.e., `x` is not equal to `X`).
`REAL UCOFST` *`Freal ucofst`*	*Input*	Unit conversion offset.
`REAL UCSCAL` *`Freal ucscal`*	*Input*	Unit conversion scale factor.
`REAL NDOFST` *`Freal ndofst`*	*Input*	Non-dimensionalization offset.
`REAL NDSCAL` *`Freal ndscal`*	*Input*	Non-dimensionalization scale factor.

`CFWREF` writes the reference and scaling data for the specified variable. (See Section 4.)

Note: To obtain conversion numbers, use the `CFUNIT` subroutine described on p. 43, or see Appendix A.

CFVINF — Get Information About a Variable

```
CALL CFVINF (STATUS, STATE, VARNAM, VARTYP, VARLEN)
Fint CF_Cvinf (status, state, varnam, vartyp, varlen)
```

INTEGER STATUS *Fint *status*	*Output*	Return status; for C it is also the function return.
INTEGER STATE *Fint *state*	*Input*	Node state of the node that contains the requested variable.
CHARACTER*(*) VARNAM *Fchar *varnam*	*Input*	Name of the variable to obtain information. It will be truncated or padded with blanks to 8 (for versions 1 and 2) or 32 (for version 3) characters as required. The name is case-sensitive (i.e., x is not equal to X).
INTEGER VARTYP *Fint *vartyp*	*Output*	The type of the requested variable. 1 = Integer 2 = Real 3 = Double precision 4 = Character
INTEGER VARLEN *Fint *varlen*	*Output*	Number of elements in the specified variable.

CFVINF returns the type and length of the specified variable. Note that if the requested variable is not present, CFVINF will not abort even if the ABORT flag is set. For character variables the length returned will be the element length times the character size.

`CFVLST` — Get List of Variables

```
CALL CFVLST (STATUS, STATE, VARLST, VARTYP, NUMVAR)
Fint CF_Cvlst (status, state, varlst, vartyp, numvar)
```

`INTEGER STATUS` *`Fint *status`*	*Output*	Return status; for C it is also the function return.
`INTEGER STATE` *`Fint *state`*	*Input*	The node state for the list of variables to be returned.
`CHARACTER*(*) VARLST(*)` *`Fchar **varlst`*	*Output*	The variables associated with the specified node.
`INTEGER VARTYP(*)` *`Fint *vartyp`*	*Output*	The types of the variables in `VARLST`. 1 = Integer 2 = Real 3 = Double precision 4 = Character
`INTEGER NUMVAR` *`Fint *numvar`*	*Input*	If equal to −1 then only return `NUMVAR`, otherwise return lists also.
	Output	The number of variables returned.

`CFVLST` returns the list of variables associated with the specified node.

CFNLST — Get List of Subnodes

```
CALL CFNLST (STATUS, STATE, NODLST, NUMNOD)
Fint CF_Cnlst (status, state, nodlst, numnod)
```

INTEGER STATUS *Fint *status*	*Output*	Return status; for C it is also the function return.
INTEGER STATE *Fint *state*	*Input*	The node state for the subnode list to be returned.
CHARACTER*(*) NODLST *Fchar **nodlst*	*Output*	List of subnodes under the specified node.
INTEGER NUMNOD *Fint *numnod*	*Input*	If equal to −1 then only return NUMNOD, otherwise return lists also.
	Output	The number of subnodes returned.

CFNLST returns a list of subnodes of the specified node.

`CFSLST` — Get List of Reference and Scaling Data

```
CALL CFSLST (STATUS, STATE, VARLST, NUMVAR)
Fint CF_Cslst (status, state, varlst, numvar)
```

`INTEGER STATUS` *`Fint *status`*	*Output*	Return status; for C it is also the function return.
`INTEGER STATE` *`Fint *state`*	*Input*	The root state of the file.
`CHARACTER*(*) VARLST` *`Fchar **varlst`*	*Output*	List of variables that have reference and scaling data.
`INTEGER NUMVAR` *`Fint *numvar`*	*Input*	If equal to −1 then only return `NUMVAR`, otherwise return lists also.
	Output	The number of variable names returned.

`CFSLST` returns a list of variables that have reference and scaling data.

CFUNIT — Common File Get Current/New UNITs

```
CALL CFUNIT (STATUS, STATE, VARNAM, MODE, CUNITS, DUNITS, CNVOFF, CNVSCL)
Fint CF_Cunit (status, state, varnam, mode, cunits, dunits, cnvoff, cnvscl)
```

INTEGER STATUS *Fint *status*	*Output*	Return status; for C it is also the function return.
INTEGER STATE *Fint *state*	*Input*	The root state of the file.
CHARACTER*(*) VARNAM *Fchar *varnam*	*Input*	The name of the variable or unit quantity for which unit/conversion data is derived.
CHARACTER*(*) MODE *Fchar *mode*	*Input*	CURRENT — get current units NEW — get conversion factors from CUNITS to DUNITS.
CHARACTER*(*) CUNITS *Fchar *cunits*	*Input*	When mode is NEW it is the current(from) units. If blank and VARNAM is a CFD variable it will be derived from reference and scaling data.
	Output	When mode is CURRENT it contains the current system of units the variable is determined to be in using the reference and scaling data for the variable.
CHARACTER*(*) DUNITS *Fchar *dunits*	*Input*	When mode is NEW it is the desired units that you wish to convert to.
	Output	When mode is CURRENT it contains the current units the variable is determined to be in using the reference and scaling data for the variable.
REAL CNVOFF *Freal *cnvoff*	*Output*	Current/new conversion offset.
REAL CNVSCL *Freal *cnvscl*	*Output*	Current/new conversion scale factor.

CFUNIT returns the current system of units and the units for any CFD variable when the mode is CURRENT. When then mode is NEW, CFUNIT returns conversion data from CUNITS to DUNITS where the input CUNITS and DUNITS can be a system of units or the actual units. If actual units are used they must conform to the format used in Appendix A, "Conversion Factors." Valid systems of units are MKS = SI = METRIC, CGS, ENGLISH = BRITISH = FSS (Foot-Slug-Second), and FPP (Foot-Pound-Second). VARNAM may be any CFD variable name or a unit type. Valid unit types are LENGTH, MASS, TIME, FORCE, DENSITY, TEMPERATURE, ENERGY, ENTROPY, VELOCITY, PRESSURE, GAS CONSTANT and VISCOSITY. If VARNAM is a CFD variable and either CUNITS is blank or mode is CURRENT then the current units will be derived from the reference and scaling data. If no reference and scaling data have been stored for the variable the default will be the MKS system. For readability the following aliases have been set up for use in the CUNITS and DUNITS variables.

```
m = meter = meters
cm = centimeter = centimeters
mm = millimeter = millimeters
ft = feet
in = inch = inches
kg = kilogram = kilograms
```

```
g = gram = grams
slug = slugs
lbf = lb = pounds
K = KELVIN
C = CENTIGRADE = CELSIUS
R = RANKINE
F = FAHRENHEIT
J = N-m
erg = dyn-cm
```

Note: These aliases can not be combined.

For consistency this subroutine should be used in all CFD applications. The following examples illustrate the use of this subroutine.

Example 1

When the variable is old and is being read into an application, the application wants the variable to be in certain units. The following reads the variable `'x'` and places it in the units of inches. This example uses the fact that `MSCALE` and `MOFF` factors are stored to convert `'x'` to the MKS system of units.

```
CALL CFRVR  (STATUS, STATE, 'x', VARLEN, 1, X)
CALL CFRREF (STATUS, STATE, 'x', MOFF, MSCALE, DOFF, DSCALE)
CALL CFUNIT (STATUS, STATE, 'x', 'NEW', 'MKS', 'in', CNVOFF, CNVSCL)
DO I = 1,VARLEN
   X(I) = ((X(I)*DSCALE + DOFF)*MSCLE + MOFF)*CNVSCL + CNVOFF
END DO
```

Note: If the data is modified and is to be stored back into the file then the inverse conversion must be performed.

Example 2

When the variable is old and is being read into an application the units that the variable is stored in may need to be known (e.g. for display). The following writes the variable `'p'` to a file and includes the units as part of the output.

```
      CALL CFRVR  (STATUS, STATE, 'p', VARLEN, 1, P)
      CALL CFRREF (STATUS, STATE, 'p', MOFF, MSCALE, DOFF, DSCALE)
      CALL CFUNIT (STATUS, STATE, 'p', 'CURRENT', CUNITS, DUNITS, CNVOFF, CNVSCL)
      CALL UTSLEN (DUNITS, LEN)
      WRITE (6, 210) 'I', 'J', 'K', 'PRESSURE'
      DO K = 1,KMAX
         DO J = 1,JMAX
            DO I = 1,IMAX
               P(I,J,K) = P(I,J,K)*DSCALE + DOFF
               WRITE (6,200) I, J , K, P(I,J,K), DUNITS(1:LEN)
            END DO
         END DO
      END DO
  200 FORMAT (3I5, F12.5, A)
  210 FORMAT (A, 5X, A, 5X, A, 7X, A)
```

Example 3

When the variable is first being stored, four factors must be saved for the variable using the subroutine CFWREF. The dimensional factors and units are known, CFUNIT is used to get the factors to convert those units to the MKS system of units. In the following the pressure is known to be in the 'FSS' system of units.

```
CALL CFUNIT (STATUS, STATE, 'p', 'NEW', 'FSS', 'MKS', MOFF,  MSCALE)
CALL CFWREF (STATUS, STATE, 'p', MOFF, MSCALE, DOFF, DSCALE)
CALL CFWVR  (STATUS, STATE, 'p', VARLEN, 1, P)
```

`CFOREC` — Common File Optimal RECord length calculation

```
CALL CFOREC (STATUS, STATE, DIMLST, NUMDIM, TYPE, PRECIS, RECLEN)
Fint CF_Corec (status, state, dimlst, numdim, type, precis, reclen)
```

`INTEGER STATUS` *`Fint *status`*	*Output*	Return status; for C it is also the function return.
`INTEGER STATE` *`Fint *state`*	*Input*	Global state of common file to search with `FULL` type.
`INTEGER DIMLST(4,NUMDIM)` *`Fint *dimlst`*	*Input*	List of zone dimensions and number of fringe points.
`INTEGER NUMDIM` *`Fint *numdim`*	*Input*	Number of dimensions in `DIMLST`.
`CHARACTER*(*) TYPE` *`Fchar *type`*	*Input*	Type of common file to compute record length (`CGD, CFL, CGF, ZDF, FULL`). For `FULL`, all variables in the CF file `STATE` are used and `DIMLST` is not used.
`INTEGER PRECIS` *`Fint precis`*	*Input*	Precision of the output variables (1=single, 2=double).
`INTEGER RECLEN` *`Fint *reclen`*	*Output*	Optimal record length.

`CFOREC` calculates the optimal record length for a common file. There are five modes '`FULL`', '`CGD`', '`CFL`', '`CGF`', or '`ZDF`'. Mode '`FULL`' works on existing common files and uses the actual variable sizes stored in the file to calulate the optimal record length. The other modes make a guess at the optimal record length based on the zone dimensions input. The '`CGD`' mode is similar to the '`CFL`' mode except that it takes into account the variable sizes of boundary conditions. If you are not storing boundary conditions then the '`CFL`' mode should be used. For version 3 files there is no record length although the record length parameter stored in the file is still used to determine the size of the conversion buffers used in the `CFRV`*`xx`* subroutines.

CFVSIZ — Common File Variable SIZe information

```
CALL CFVSIZ (STATUS, STATE, MAXI, MAXR, MAXD)
Fint CF_Cvsiz (status, state, maxi, maxr, maxd)
```

INTEGER STATUS *Fint *status*	*Output*	Return status; for C it is also the function return.
INTEGER STATE *Fint *state*	*Input*	Global state of common file to search.
INTEGER MAXI *Fint *maxi*	*Output*	The size of the largest integer variable in the file.
INTEGER MAXR *Fint *maxr*	*Output*	The size of the largest real variable in the file.
INTEGER MAXD *Fint *maxd*	*Output*	The size of the largest double precision variable in the file.

CFVSIZ searches the entire file and locates the largest variable of the given type. This is useful for check and/or allocating memory for reading file variables.

`CFGTRT` — Get File Root State from Any File State

```
CALL CFGTRT (STATUS, STATE, ROOTSTATE)
Fint CF_Cgtrt (status, state, rootstate)
```

`INTEGER STATUS` *`Fint *status`*	*Output*	Return status; for C it is also the function return.
`INTEGER STATE` *`Fint *state`*	*Input*	Any state of common file.
`INTEGER ROOTSTATE` *`Fint *rootstate`*	*Output*	Root state of common file.

`CFGTRT` returns the root state of a common file, given any state in the file.

`CFHOST` — Common File get HOST type

```
CALL CFHOST (CPU, OS)
void CF_Chost (cpu, os)
```

`INTEGER CPU` *`Fint *cpu`*	*Output*	Output cpu binary type, where: 0 = Unknown 1 = Cray (1, 2, X-MP, Y-MP, C90) 2 = AX, F_FLOATING and D_FLOATING (non-AXP native) 3 = IBM (360/370) 4 = CONVEX (C1, C2 native mode) 5 = IEEE big endian (IRIS 2000/3000/4D) 6 = IEEE little endian 7 = VAX, F_FLOATING and G_FLOATING (AXP native) 8 = Unused 9 = Unused
`INTEGER OS` *`Fint *os`*	*Output*	Output operating system type, where: 0 = Unknown 1 = Cray Unix 2 = Vax VMS 3 = ALPHA Unix 4 = SUN Unix 5 = RS6K Unix 6 = CONVEX Unix 7 = SGI Unix 8 = HP Unix 9 = PARAGON Unix

6 Status Codes

All of the common file library routines return a status code through the first argument to indicate whether the routine failed or succeeded. Below is a list of all of the status codes.

Table 1: Status Codes

Code	Meaning
< 0	ADF core error message (see ADF document for details).
0	Successful completion (no errors encountered).
100	File header validation error.
101	Node header validation error.
102	File header indicates `IMPORT` did not complete. Rerun `IMPORT`.
200	File was never opened.
201	File was never closed.
202	Invalid mode supplied to `CFOPEN` (must be `OLD`, `NEW`, or `SCRATCH`).
300	Requested variable does not exist.
301	Invalid variable name (cannot be blank).
302	Unable to allocate a new variable (out of variable space).
303	Variable is not of the proper type.
304	Invalid record length.
305	Invalid record number.
306	Invalid operation for C I/O.
400	Requested node does not exist.
401	Invalid node name (cannot be blank).
402	Unable to allocate a new node (out of node space).
403	Requested node already exists.
500	Invalid `STATE`.
501	Maximum number of files are already open.
502	Unknown `STATE`.
504	Error converting C to Fortran string.
505	Error converting Fortran to C string.
600	Invalid parameter name.
601	Invalid unit number.
602	Invalid record length.
603	Invalid machine code.
604	Invalid maximum number of nodes.
605	Invalid maximum number of variables.
606	Invalid maximum number of integer elements for node headers.
607	Invalid maximum number of real elements for node headers.
608	Invalid maximum number of character elements for node headers.
609	Invalid dimensions for record length calculations.

Continued on next page

Table 1: Status Codes (*Continued*)

Code	Meaning
610	Invalid precision for record length calculations.
611	Invalid common file version.
650	File open, cannot change unit number.
651	File open, cannot change record length.
652	File open, cannot change maximum number of nodes.
653	File open, cannot change maximum number of variables.
654	File open, cannot change maximum number of integer elements.
655	File open, cannot change maximum number of real elements.
656	File open, cannot change maximum number of character elements.
700	Invalid mode in `CFUNIT` (must be '`NEW`' or '`CURRENT`').
701	Unknown desired units type
702	Unknown current units type
801	Invalid operation for the files common file version
900	Fortran `OPEN` error.
901	Fortran `CLOSE` error.
902	Fortran `READ` error.
903	Fortran `WRITE` error.
904	Conversion to or from this machine is not supported.
905	Error allocating memory.

7 Default Library Values and Limits

The current implementation of the common file library has the following set of maximum and default values. Note that the values are applied at the time of file creation.

Table 2: Default Library Values and Limits

Parameter	Maximum			Default	Description
	Vers 1	Vers 2	Vers 3		
ABORT	N/A	N/A	N/A	1	Abort-on-error flag
MAX_NODES	128	512	none	64	Maximum number of subnodes per node
MAX_VARIABLES	128	512	none	64	Maximum number of variables per node
MAX_INTEGERS	128	none	none	64	Maximum number of INTEGER elements in a node header
MAX_REALS	128	none	none	64	Maximum number of REAL elements in a node header
MAX_CHARACTERS	4	none	none	2	Maximum number of CHARACTER*80 elements in a node header
MAX_FILES	7	7	15	max	Maximum number of common files that can be open simultaneously by the application. This cannot be altered by the application.

8 Definitions for CFD Applications

This section describes the conventions to be used when using a common file for CFD applications. Consistent application of these conventions will assure that data will be accessible to a developing base of pre- and post- processing tools. For a picture representation of the following definitions see Figure 1 at the end of this section.

8.1 Node Identifiers

8.1.1 Zone Node Identifiers

Zone nodes are subnodes of the root node. A zone node is accessed and the header read through the following Fortran code fragment:

```
CHARACTER*32 ZNAME
INTEGER ZONE
INTEGER STATUS, RSTATE, ZSTATE

! Note that the default sizes of IPARZ, FPARZ and CPARZ are used.

INTEGER IPARZ(64)
REAL    FPARZ(64)
CHARACTER*80 CPARZ(2)

! RSTATE is the state of the root node returned by CFOPEN
! All zone nodes under the root node are named 'ZONE nnn'

ZONE = zone number (1, 2, 3, etc.)
WRITE (ZNAME, '(''ZONE '', I3)') ZONE
CALL CFSNOD (STATUS, RSTATE, ZNAME, ZSTATE)
CALL CFRNOD (STATUS, ZSTATE, IPARZ, FPARZ, CPARZ)
```

After executing the above code, you may access the data for the zone through the state variable ZSTATE.

8.1.2 Boundary Node Identifiers

Boundary nodes are subnodes of a zone node. A boundary node is accessed through the following Fortran construct:

```
CHARACTER*32 BNAME
INTEGER BNDRY
INTEGER STATUS, BSTATE, ZSTATE

! All boundary nodes for a given zone are stored under the zone
! node with a name of 'BNDRY n' where n is a number defined below.

BNDRY = boundary (1=I1, 2=IMAX, 3=J1, 4=JMAX, 5=K1, 6=KMAX, 7=Chimera,
                  ≥ 8=Unstructured)
WRITE (BNAME, '(''BNDRY  '', I3)') BNDRY
CALL CFSNOD (STATUS, ZSTATE, BNAME, BSTATE)
```

8.1.3 Interior Node Identifiers

Interior nodes are subnodes of a zone node (for 3-D unstructured grids only). An interior node is accessed through the following Fortran construct:

```
CHARACTER*32 INAME
INTEGER STATUS, ISTATE, ZSTATE

! An interior node for a given zone is stored under the zone
! node with the name 'INTERIOR'.

INAME = 'INTERIOR'
CALL CFSNOD (STATUS, ZSTATE, INAME, ISTATE)
```

8.2 Node Header Definitions

Note: All dimensional values stored in the common sections of the node headers must be in SI units. Data stored in the application-specific sections can be in any units the application desires.

8.2.1 Root Node Common Definitions

For unstructured grids, `IPAR(1)`, `IPAR(2)`, and `IPAR(3)` are not used. For a file containing both structured and unstructured zones, these parameters will have the correct values for the structured zones.

`IPAR(1)`	Maximum `I` in all zones
`IPAR(2)`	Maximum `J` in all zones
`IPAR(3)`	Maximum `K` in all zones
`IPAR(4)`	Number of zones
`IPAR(5)`	Maximum number of points in any one zone
`IPAR(6)`	Symmetry flag: 0 = no symmetry assumed, 1 = x-y axi-symmetric, 2 = y-z axi-symmetric, 3 = x-z axi-symmetric
`IPAR(7)`	Mxset
`IPAR(8)`	Rotation system flag: 0 = none, 1 = rotating system, 2 = gravity system
`IPAR(9-39)`	Reserved for future use
`IPAR(40-64)`	Reserved for application-specific data

`FPAR(1)`	p_0, freestream stagnation pressure
`FPAR(2)`	T_0, freestream stagnation temperature
`FPAR(3)`	a_0, freestream stagnation speed of sound
`FPAR(4)`	ρ_0, freestream stagnation density
`FPAR(5)`	M, freestream Mach number
`FPAR(6)`	p, freestream static pressure
`FPAR(7)`	T, freestream static temperature

FPAR(8)	a, freestream speed of sound
FPAR(9)	ρ, freestream density
FPAR(10)	k, freestream k of k-ϵ or k-ω turbulence model
FPAR(11)	ϵ or ω, freestream ϵ or ω of k-ϵ or k-ω turbulence model
FPAR(12)	μ, freestream viscosity
FPAR(13)	Re, Reynolds number
FPAR(14)	α, angle of attack
FPAR(15)	β, yaw angle
FPAR(16-23)	Reserved for future use
FPAR(24)	β_∞, effective γ
FPAR(25)	R, gas constant
FPAR(26)	γ, ratio of specific heats (1.4)
FPAR(27)	Pr, Prandtl number (0.72)
FPAR(28)	Pr_t, turbulent Prandtl number (0.9)
FPAR(29)	x, x base point of axi-symmetric line
FPAR(30)	y, y base point of axi-symmetric line
FPAR(31)	z, z base point of axi-symmetric line
FPAR(32)	M, slope of axi-symmetric line
FPAR(33)	A, angle of rotation about axi-symmetric line
FPAR(34-36)	Rotation system xyz center (IPAR(8) = 1)
FPAR(37-39)	Rotation system xyz rotation rate (IPAR(8) = 1), or gravity system xyz terms (IPAR(8) = 2)
FPAR(40-64)	Reserved for application-specific data
CPAR(1)	Grid title
CPAR(2)	Flowfield title

8.2.2 Wind-US Root Node Application Specific Data

IPAR(59)	Last time level completed (Global Newton)
IPAR(60)	-1 for new format global header
IPAR(61)	Last I plane completed (marching)
IPAR(62)	Last zone completed
IPAR(63)	Number of k-ϵ cycles
IPAR(64)	Number of cycles
FPAR(60)	Global Newton big norm
FPAR(61)	Global Newton L2 norm
FPAR(62)	Global Newton max convergence variable ever

8.2.3 Zone Node Header Common Definitions

Zone nodes can contain either structured or unstructured grids. These two types are distinguished based on IPAR(9) in the Zone Node Header. IPAR(9) will be 0 for structured grids and 1 or 2 for unstructured grids.

Structured Grid

`IPAR(1)`	`I` dimension
`IPAR(2)`	`J` dimension
`IPAR(3)`	`K` dimension
`IPAR(4)`	Number of fringe points
`IPAR(5)`	Number of overlapping tracking definitions
`IPAR(6)`	Size of overlapping definition work array
`IPAR(7)`	Reserved for future use
`IPAR(8)`	Rotation flag: 0 = Non-rotating; 1 = Rotating (see `FPAR(11-16)`)
`IPAR(9)`	Grid type: 0 = Structured; 1,2 = Unstructured
`IPAR(10)`	Gas model: 0 = Ideal gas 1 = Thermally perfect (frozen chemistry) 2 = Equilibrium air 3 = Finite rate
`IPAR(11)`	Turbulence model: 0 = Euler (Inviscid) 1 = Laminar 2 = Baldwin-Lomax 3 = Cebeci-Smith 4 = k-ϵ (obsolete) 5 = Baldwin-Lomax and PDT 6 = Baldwin-Barth (1 equation) 7 = Spalart-Allmaras (1 equation) 8 = SST-Menter (2 equation, k-ω) 10 = Chien k-ϵ
`IPAR(12)`	Cell/node/variable relationships 0 = Variable values are at node points 1 = Variable values are at cell centers
`IPAR(13)`	Wall function mode 0 = No wall function 1 = White/Christoph model
`IPAR(20)`	Boundary type for `I = 1` boundary 0 = Undefined 1 = Reflection/symmetry 2 = Adiabatic wall 3 = Freestream 4 = Viscous wall 5 = Unused 6 = Unused 7 = Arbitrary inflow 8 = Outflow 9 = Inviscid wall 10 = Self-closing 11 = Singular axis 12 = Inviscid axis and wall (not used in Wind-US) 13 = Coupled/point by point 14 = Unused 15 = Bleed 16 = Pinwheel axis

	17 = Frozen
	18 = Chimera
`IPAR(21)`	Boundary type for `I = IMAX` boundary
`IPAR(22)`	Boundary type for `J = 1` boundary
`IPAR(23)`	Boundary type for `J = JMAX` boundary
`IPAR(24)`	Boundary type for `K = 1` boundary
`IPAR(25)`	Boundary type for `K = KMAX` boundary
`IPAR(26)`	Grid velocity flag; 0 = none, 1 = global, 13 = point by point
`IPAR(27-39)`	Reserved for future use
`IPAR(40-64)`	Reserved for application-specific data
`FPAR(1-7)`	Zone min/max (x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}, checksum)
`FPAR(8-10)`	Global translation velocity (u, v, w)
`FPAR(11-13)`	Global rotation velocity (R, Θ, Ψ)
`FPAR(14-16)`	Global rotation center (x, y, z)
`FPAR(17-39)`	Reserved for future use
`FPAR(40-64)`	Reserved for application-specific data
`CPAR(1)`	Zone title
`CPAR(2)`	Characters 1:40, grid generator identification
	Characters 41:80, flow solver identification

Unstructured Grid

`IPAR(1)`	Number of points
`IPAR(2)`	Total number of edges, including internal as well as boundary edges. (Optional; only needed if edge data structure is stored.)
`IPAR(3)`	Total number of faces, including internal as well as boundary faces.
`IPAR(4)`	Number of cells (0 $\Rightarrow$ surface)
`IPAR(5)`	Max number of nodes per face
`IPAR(6)`	Max number of faces per cell
`IPAR(7)`	Number of surface offset data entries
`IPAR(8)`	Max number of nodes per cell
`IPAR(9)`	Grid type:
	0 = Structured
	1 = Unstructured (tetrahedral)
	2 = Hybrid (mixed element types)
`IPAR(10)`	Gas model:
	0 = Ideal gas
	1 = Thermally perfect (frozen chemistry)
	2 = Equilibrium air
	3 = Finite rate
`IPAR(11)`	Turbulence model:
	0 = Euler (Inviscid)
	1 = Laminar
	2 = Baldwin-Lomax
	3 = Cebeci-Smith
	4 = k-ϵ
	5 = Baldwin-Lomax and PDT
`IPAR(12)`	Cell/node/variable relationships:
	0 = Variable values are at node points

	1 = Variable values are at cell centers
IPAR(13)	Number of fringe points
IPAR(14)	Number of sequenced cells
IPAR(15-18)	Reserved for flow code use
IPAR(19)	Total number of elements (CGNS)
IPAR(20)	Number of boundary points
IPAR(21)	Number of boundary edges
IPAR(22)	Number of boundary faces
IPAR(23)	Boundary condition type (also used to distinguish between 2-D and 3-D grids): 0 = Face (3D) 1 = Node (3D) 2 = Edge (2D) 3 = Node (2D)
IPAR(24)	Number of closed boundaries (Curves-2D, Surfaces-3D) up to a maximum of 20
IPAR(25-58)	Array of boundary size pairs: (25,27,...) - No. of points (2D or 3D surface) (26,28,...) - No. of faces (Full 3D boundary)
IPAR(60)	Number of overlapping tracking definitions
IPAR(61)	Size of overlapping definition work array
FPAR(1-39)	Reserved for future use
FPAR(40-64)	Reserved for application-specific data
CPAR(1)	Zone title
CPAR(2)	Characters 1:40, grid generator identification Characters 41:80, flow solver identification

8.2.4 Wind-US Zonal Node Application Specific Data

IPAR(55)	Compressor face mode
IPAR(56)	I location of downstream pressure
IPAR(57)	J location of downstream pressure
IPAR(58)	K location of downstream pressure
IPAR(59)	Downstream pressure variable flag (I, J, or K)
IPAR(63)	Unstructured grid type: 0 = Mixed cells 1 = Tets only 5 = Pyramid only 11 = Prism only 111 = Hex only
FPAR(37-39)	*.dat* file SMO1-3 smoothing parameters
FPAR(40-42)	Current SMO1-3 levels in solution file
FPAR(43-54)	Load convergence FPxyz, FVxyz, MPxyz, MVxyz
FPAR(55)	Compressor face T0
FPAR(56)	Compressor face mass flow
FPAR(57)	Compressor face Mach
FPAR(58)	Downstream pressure for IJK mode
FPAR(59)	Downstream Mach for IJK mode
FPAR(60)	

FPAR(61)	Downstream pressure
FPAR(62)	Mass flow ratio
FPAR(63)	
FPAR(64)	Capture area used with specified mass flow ratio and bleed rate boundary conditions

8.2.5 Boundary Node Application Specific Data

IPAR(1)	BC format: 0 = old, 1 = new
IPAR(8)	Unstructured offset (CGNS)
IPAR(20)	BC code (CGNS)
IPAR(40)	Wall function mode 0 = No wall function 1 = White/Christoph model
IPAR(58)	Boundary rotation mode
IPAR(59-64)	Rotation specific data depending on mode
FPAR(40)	Capture area (see zonal FPAR(64))
FPAR(41)	Bleed area (see BLAREA variable in Section 8.3.6)
CPAR(1)	Capture area region name
CPAR(2)	Bleed area region name

8.3 Variable Identifiers

8.3.1 Geometry

x	x, x coordinate
y	y, y coordinate
z	z, z coordinate
IBLANK	Blanking data asscociated with overlapping grids
ui	x direction grid velocity
vi	y direction grid velocity
wi	z direction grid velocity

The variables x, y, and z are located under the zone node for structured grids. For unstructured grids, x, y, and z for the boundary points are located under the zone node and x, y, and z for the interior points (3D) are located under the interior node. Note that IBLANK has labels encoded in it as follows: 1 = a normal point, $-(label+1)$ = a hole point, and $+(label+1)$ = a fringe point, where *label* is a positive number.

The following variables are applicable to unstructured grids only, and are located under the zone node for boundary surfaces and under the interior node for volumes. These variables describe the connectivity of the faces and/or edges.

facp*i*	Index for point i of a face, $i = 1$ to the number of points per face. facp4 = 0 for triangular faces in a mixed quad/tri face grid. The facp values for a particular face should be oriented such that the face normal points into the domain.
edgp*i*	Index for point i of an edge segment
face*i*	Index for edge i of a face, $i = 1$ to the number of edges per face

The `edgpi` and `facei` variable definitions are provided for future extensions and will not be initially supported by common applications (such as post-processing).

The following variables are applicable to unstructured grids only, and are located under the the interior node. These variables describe the connectivity of the cells.

`celpi`	Index for point i of a cell, $i = 1$ to the number of points per cell. `celpi` $= 0$ for nodes of dimensionality greater than the current cell. (I.e., `celp5` $= 0$ for a tetrahedral cell.)
`celfi`	Index for face i of a cell, $i = 1$ to the number of faces per cell
`celtyp`	The type of cell (see definition of `IPAR(63)` in Section 8.2.4). Used for hybrid unstructured grids only.

The `celfi` variable definitions are provided for future extensions and will not be initially supported by common applications (such as post-processing)

The following variables are applicable to unstructured grids only, and are located under the zone node. These variables describe collections of faces or surfaces. Currently, each surface must contain only a single face type (i.e., all triangles or all quads). Each of these variables are of size `nsurfs` = `ipar(7)`.

`srfoff`	Starting index in the face list of a surface segment
`srfsiz`	Number of faces on a surface segment
`srfid`	ID number for a surface segment
`srfbc`	Boundary condition for a surface segment, BC codes same as structured grid
`srfvrt`	For hybrid grids, the number of points per face for all faces of that boundary surface
`srfpts`	Number of points referenced by all of the faces on that boundary surface

8.3.2 General Flow Variables

`rho`	ρ, density
`rho*u`	ρu, x component of momentum per unit volume
`rho*v`	ρv, y component of momentum per unit volume
`rho*w`	ρw, z component of momentum per unit volume
`rho*e0`	ρe_0, stagnation energy per unit volume
`P`	p, pressure
`T`	T, temperature
`u`	u, x component of velocity
`v`	v, y component of velocity
`w`	w, z component of velocity
`e0`	e_0, stagnation energy per unit mass
`M`	M, Mach number
`s`	s, entropy
`h`	h, enthalpy
`omegax`	ω_x, x component of vorticity
`omegay`	ω_y, y component of vorticity
`omegaz`	ω_z, z component of vorticity

For rotating systems (`IPAR(8)` $= 1$), an "`r`" is appended to the name of the following variables to indicate the variable is in a rotating reference system: `rho*u`, `rho*v`, `rho*w`, `rho*e0`, `u`, `v`, `w`, `e0`, `M`.

8.3.3 Turbulence Variables

`mul`	μ_l, laminar viscosity
`mut`	μ_t, turbulent viscosity
`k`	k, kinetic energy (k-ϵ model)
`rho*k`	ρk (k-ϵ model)
`epsilon`	ϵ, rate of dissipation (k-ϵ model)
`rho*epsi`	$\rho\epsilon$ (k-ϵ model)
`K`	k (k-ϵ and SST models)
`omega`	Ω (SST model)
`anut`	η_t (Baldwin-Barth and Spalart-Allmaras models)

8.3.4 Chemistry Variables

`a`	a, local speed of sound
`beta`	β, effective gamma
`Z`	Z, compressibility
`kappa`	κ, thermal conductivity
`etake`	$\nu_{K.E.}$, kinetic energy efficiency
`PHI`	Mass fraction of non-reacting species
`H`	Mass fraction of H
`N`	Mass fraction of N
`O`	Mass fraction of O
`H2`	Mass fraction of H_2
`N2`	Mass fraction of N_2
`O2`	Mass fraction of O_2
`OH`	Mass fraction of OH
`NO`	Mass fraction of NO
`H2O`	Mass fraction of H_2O
`CO2`	Mass fraction of CO_2
`rho*H`	ρH, ρ times the mass fraction of H; similarly for N, O, etc.

8.3.5 Miscellaneous Variables

`Cp`	C_p, pressure coefficient
`delta*`	δ^*, displacement thickness of boundary layer
`THETA`	θ, momentum thickness of boundary layer
`Redelta*`	Re_{δ^*}, Reynolds number based on δ^*
`Cf1`	C_{f1}, skin friction in `I` direction
`Cf2`	C_{f2}, skin friction in `J` direction
`Cf3`	C_{f3}, skin friction in `K` direction

8.3.6 Wind-US Application Specific Variables

The following variables are used by Wind-US. For the new BC format, the "`_n`" variables are for node-centered grids, and the "`_c`" variables are for cell-centered grids. For cell vertex grids the size of the boundary is the number of points, and for cell-centered grids it is the number of faces.

Table 3: Wind-US Application Specific Variables

Variable	Description	Location	Type
`maxr`	Max residual for all zones for each equation	Root	Real
`BLAREA`	Bleed area	Root	Real
`SETDATA(3,*)`	For each set; `(1,*)` = # iterations, `(2,*)` = max residual, and `3` = integrated time (located under zone node)	Zone	Real
`LABLST`	Label for overlapping tracking	Zone	Integer
`LABTYP`	Hole/fringe type for overlapping tracking	Zone	Integer
`LABIND`	Index into `LABDEF` for overlapping tracking	Zone	Integer
`LABDEF`	Generation data for overlapping tracking	Zone	Real
`NZN`	If > 0, zone coupled to; if ≤ 0, boundary condition	Boundary	Integer
`NBD`	Boundary coupled to	Boundary	Integer
`IFRG`	The I value of the fringe point (`FRNGBND`)	Boundary	Integer
`JFRG`	The J value of the fringe point (`FRNGBND`)	Boundary	Integer
`KFRG`	The K value of the fringe point (`FRNGBND`)	Boundary	Integer
`I1`	Node coupled to (first coordinate)	Boundary	Integer
`I2`	Node coupled to (second coordinate)	Boundary	Integer
`I3`	Node coupled to (third coordinate)	Boundary	Integer
`F1`	First coordinate tri/bilinear interpolating factor	Boundary	Real
`F2`	Second coordinate tri/bilinear interpolating factor	Boundary	Real
`F3`	Third coordinate trilinear interpolating factor	Boundary	Real
`Zone_[nc]`	If > 0, zone coupled to; if ≤ 0, boundary condition	Boundary	Integer
`BndryNo_[nc]`	The number of the boundary coupled to	Boundary	Integer
`Cell_n,Cell_c`	The cell number containing the coupled point/face. For structured grids it is the IJK product for the point.	Boundary	Integer
`CCell_n,CCell_c`	The cell number in the coupled zone containing the coupled point/face.	Boundary	Integer
`CellLoc_n1,2,3`	For structured grids, the IJK location plus the bilinear factor in the coupled-to cell. For unstructured grids, the 1,2,3 weighting factor of the 1,2,3 boundary face point. For quads the fourth weighting factor is $1.0-\text{sum}(1,2,3)$.	Boundary	Double
`CellLoc_c1,2,3`	For structured grids, the IJK location plus the bilinear factor in the coupled-to cell. For unstructured grids, the X,Y,Z value of the cell center of the coupled face.	Boundary	Double
`CFace_c`	Face used to parameterize XYZ in `CellLoc_c`	Boundary	Integer
`Trans`	Turbulent transition specification array	Boundary	Real
`Temp`	Temperature specification array, K	Boundary	Real

The variables `Trans` and `Temp` are stored in boundary nodes in the *.cfl* file since they are solution-specific data.

For hybrid grids we have the following. When an unstructured cell-centered grid is coupled to a structured vertex-centered grid, the `CellLoc_c` data contains the structured bilinear factors used by the structured zone to interpolate the data to the unstructured face-centered location. For the

other way, `CellLoc_n` contains the parameterized value of the structured point being coupled. The parameterization is based on the unstructured zone's face that is coupled to. This face number is stored in the `CFace_n` array. This is done since this face is not unique. Note this is not a problem for cell-centered unstructured-to-unstructured, since the parameterization is based on the unstructured face being coupled, and so there is no ambiguity. The follow code segments define the parameterization of the XYZ values into `CellLoc_`*x*.

The following takes the parameterization back to XYZ:

```
!-----  Copy out the parameterization
        e123(1:3) = paramxyz(n,1:3)

!-----  Calculate the coordinate vectors
        if ( VrtcsPerBndryFace >= 4 .and. facep(4,n) > 0 ) then
!-----    Quad face
          v123(1,1:3) = VrtxXyz(facep(3,n),1:3) - VrtxXyz(facep(1,n),1:3)
          v123(2,1:3) = VrtxXyz(facep(4,n),1:3) - VrtxXyz(facep(2,n),1:3)
        else
!-----    Tri face
          v123(1,1:3) = VrtxXyz(facep(2,n),1:3) - VrtxXyz(facep(1,n),1:3)
          v123(2,1:3) = VrtxXyz(facep(3,n),1:3) - VrtxXyz(facep(1,n),1:3)
        end if
        norm(1)   = (v123(1,2)*v123(2,3) - v123(2,2)*v123(1,3))
        norm(2)   = (v123(1,3)*v123(2,1) - v123(2,3)*v123(1,1))
        norm(3)   = (v123(1,1)*v123(2,2) - v123(2,1)*v123(1,2))
        v123(3,:) = norm / sqrt(sum(norm**2)) * sqrt(sum(v123(1,:)**2))

!-----  Calculate the xyz cell center
        xyz(n,1) = VrtxXyz(facep(1,n),1) + sum(e123*v123(:,1))
        xyz(n,2) = VrtxXyz(facep(1,n),2) + sum(e123*v123(:,2))
        xyz(n,3) = VrtxXyz(facep(1,n),3) + sum(e123*v123(:,3))
```

The following parameterizes XYZ based on the input face:

```
!-----  Setup the equations
        if ( VrtcsPerBndryFace >= 4 .and. facep(4) > 0 ) then
!-----    Quad face
          v123(1,1:3) = VrtxXyz(1:3,facep(3)) - VrtxXyz(1:3,facep(1))
          v123(2,1:3) = VrtxXyz(1:3,facep(4)) - VrtxXyz(1:3,facep(2))
        else
!-----    Tri face
          v123(1,1:3) = VrtxXyz(1:3,facep(2)) - VrtxXyz(1:3,facep(1))
          v123(2,1:3) = VrtxXyz(1:3,facep(3)) - VrtxXyz(1:3,facep(1))
        end if
        norm(1)   = (v123(1,2)*v123(2,3) - v123(2,2)*v123(1,3))
        norm(2)   = (v123(1,3)*v123(2,1) - v123(2,3)*v123(1,1))
        norm(3)   = (v123(1,1)*v123(2,2) - v123(2,1)*v123(1,2))
        v123(3,:) = norm / sqrt(sum(norm**2)) * sqrt(sum(v123(1,:)**2))
        b123 = xyzc - xyz(:,facep(1))

!-----  Use Cramer's rule to solve the equations Ax = B
        call bg_ggutil_linsolve3x3_f ( status, v123, b123, e123 )
        if (status /= 0 ) e123 = 0
```

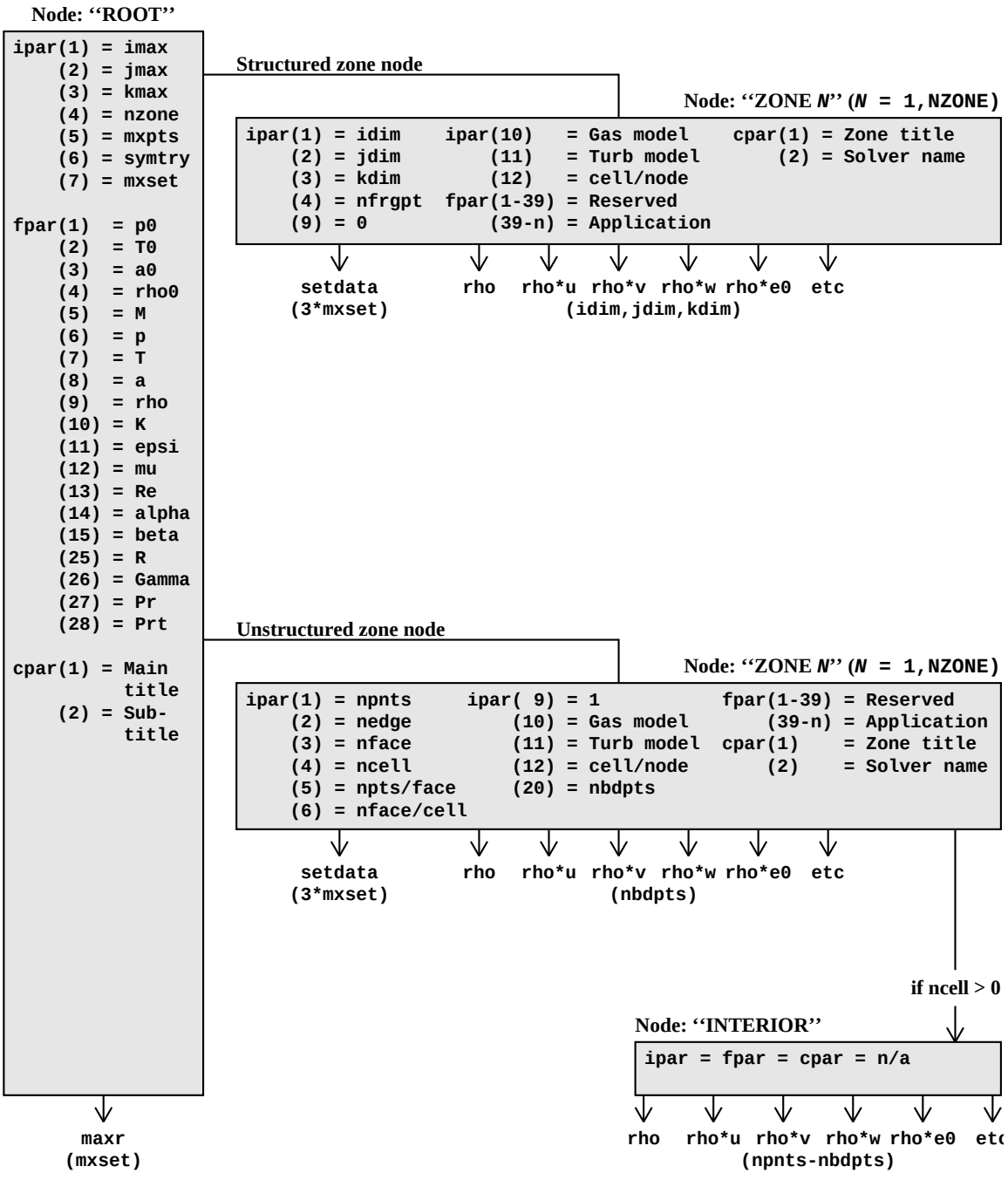

Figure 1: Common File Layout for *.cfl* Files

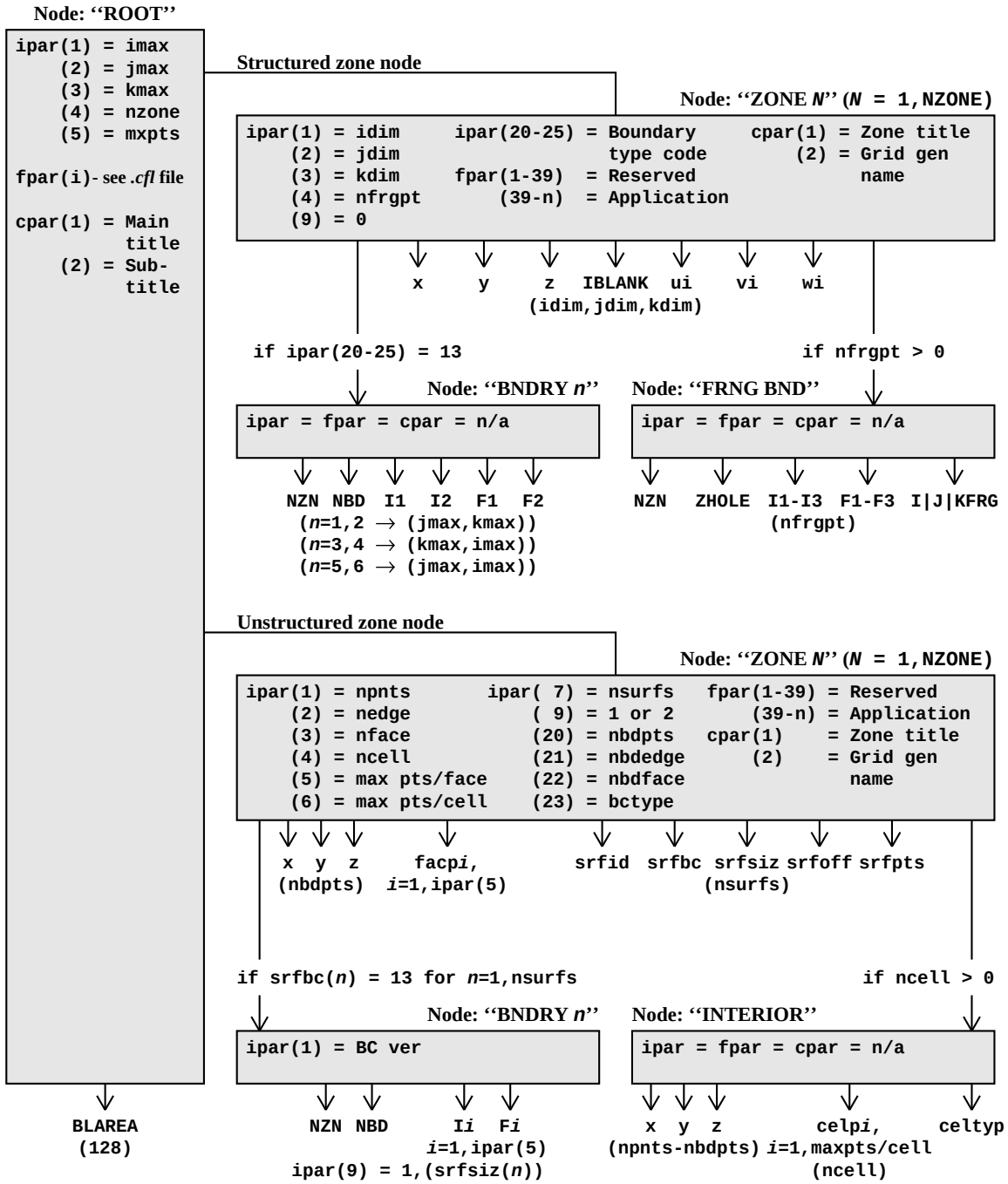

Figure 2: Common File Layout for *.cgd* Files (Old BC Format)

Appendix A. Conversion Factors

This section includes factors for converting to and from SI units. These factors are returned from the subroutine CFUNIT. Only the main factors are included here for reference (CFUNIT will return conversion factors from any supported units to any other supported units of the same type e.g. _C to _R) and to illustrate how basic unit symbols are combined. The following sections begin with the unit type and a list of valid variable names for that type (if any), followed by a list of conversions from the units on the left to the units on the right. These units are used in the CUNITS and DUNITS variable when the mode is 'NEW' and are returned in DUNITS when the mode is 'CURRENT'.

Mass

kg	×	0.0685217658567918	=	slug
slug	×	14.5939029372064	=	kg
kg	×	2.20462262184878	=	lb_m
lb_m	×	0.453592370000000	=	kg
slug	×	32.1740485564304	=	lb_m
lb_m	×	0.0310809501715673	=	slug
kg	×	1000.	=	g
g	×	0.001	=	kg

Length, x, y, z

m	×	3.28083989501312	=	ft
ft	×	0.304800000000000	=	m
m	×	39.3700787401575	=	in
in	×	0.0254	=	m
m	×	100.0	=	cm
cm	×	0.01	=	m
m	×	1000.0	=	mm
mm	×	0.001	=	m

Velocity, u, v, w

m/s	×	3.28083989501312	=	ft/s
ft/s	×	0.3048	=	m/s
m/s	×	39.3700787401575	=	in/s
in/s	×	0.0254	=	m/s
m/s	×	100.0	=	cm/s
cm/s	×	0.01	=	m/s
m/s	×	1000.0	=	mm/s
mm/s	×	0.001	=	m/s

Force

N	×	0.224808943099710	=	lb_f
lb_f	×	4.44822161526050	=	N
N	×	1×10^{5}	=	dyn
dyn	×	1×10^{-5}	=	N

Pressure, `p`

N/m^2	×	0.0208854342331501	=	lb_f/ft^2
lb_f/ft^2	×	47.8802589803358	=	N/m^2
N/m^2	×	0.00014503773773020 9	=	lb_f/in^2
lb_f/in^2	×	6894.75729316836	=	N/m^2

Gas constant, `R`

m^2/s^2-K	×	5.97995038991999	=	ft^2/s^2-R
ft^2/s^2-R	×	0.167225467570038	=	m^2/s^2-K

Viscosity, `mul`, `mut`

kg/m-s	×	0.0208854342331501	=	slug/ft-s
slug/ft-s	×	47.8802589803358	=	kg/m-s

`k` (*of* k-ϵ)

m^2/s^2	×	10.7639104167097	=	ft^2/s^2
ft^2/s^2	×	0.09290304	=	m^2/s^2

`epsilon` (*of* k-ϵ)

m^2/s^3	×	10.7639104167097	=	ft^2/s^3
ft^2/s^3	×	0.09290304	=	m^2/s^3

`rho*k` (*of* ρk-ϵ)

kg/m-s^2	×	0.0208854342331501	=	slug/ft-s^2
slug/ft-s^2	×	47.8802589803358	=	kg/m-s^2

`rho*epsi` (*of* ρk-ϵ)

kg/m-s^3	×	0.0208854342331501	=	slug/ft-s^3
slug/ft-^{3}	×	47.8802589803358	=	kg/m-s^3

Thermal conductivity, `kappa`

N/s-K	×	0.124893860586174	=	lb_f/s-R
lb_f/s-R	×	8.00679869536116	=	N/s-K

Density, `rho`

kg/m^3	×	0.00194032033197972	=	slug/ft^3
slug/ft^3	×	515.378818393196	=	kg/m^3

`rho*u`, `rho*v`, `rho*w`

kg/m^2-s	×	0.00636588035426416	=	slug/ft^2-s
slug/ft^2-s	×	157.087463846246	=	kg/m^2-s

`rho*e0`

kg/m-s^2	×	0.0208854342331501	=	slug/ft-s^2
slug/ft-s^2	×	47.8802589803358	=	kg/m-s^2

Energy, `e`, `h`

J (= N-m)	×	0.737562149277265	=	lb_f-ft
lb_f-ft	×	1.3558179483314	=	J (= N-m)
J	×	1×10^7	=	erg
erg	×	1×10^{-7}	=	J

Entropy, `s`

J/K (= N-m/K)	×	0.409756760453328	=	lb_f-ft/R
lb_f-ft/R	×	2.44047224234608	=	J/K (= N-m/K)

Temperature, `T`

K	×	9/5	=	R
R	×	5/9	=	K

Note: Conversion factors and reference conditions are stored in single precision format. When using the conversion factors in computations, use double precision calculations and assign to a single precision variable. This will ensure that accurate conversion factors and reference conditions will be maintained. Failure to do this may cause slight differences in the residual history.

Appendix B. Transferring Common Files Between Computer Systems

Common files created on CONVEX, IBM RS/6000, IRIS, VAX, VAX ALPHA, HP, and PARAGON systems can be freely interchanged among each other as the common file library routines automatically recognize the type of machine on which the file was created and, if necessary, perform the appropriate conversions without user intervention. For version 3 files the VAX mainframes are not supported. Obviously, this conversion has some cost. If a file is moved to a machine that is of a different type than the creating machine and is going to be repeatedly processed on that machine, it may be beneficial to "import" the common file using *cfcnvt*.

If a common file is going to be moved between a Cray and any of the other supported systems, then the file must be converted with the CFCRAY utility described in Appendix C. This conversion is necessary due to the differing word size of the Cray machine. Version 3 files may be interchanged freely between all supported platforms including the Cray.

In all cases, remember that common files are binary files. When using file transfer programs like *ftp*, remember to use the *binary* or *image* mode when transferring common files.

Appendix C. CFCRAY — Convert Version 1 or 2 Files to and from Cray Format

CFCRAY is a common file utility that converts a common file in CONVEX, VAX ALPHA, VAX, IRIS, HP, PARAGON or IBM RS/6000 format into Cray format and vice-versa. This utility runs only on the Cray. This program is not used for version 3 files.

C.1 Converting Version 1 or 2 Files to Cray Format

To convert a file to Cray format, it must first be transferred to the Cray machine using *ftp* (binary mode), *rcp* or some other binary transfer mechanism. Then invoke CFCRAY by entering cfcray and fill in the prompts as follows:

```
% cfcray
Enter input  file name (<CR> to quit): vax_grid.cgd
Enter output file name               : cray_grid.cgd
1=Cray
2=VAX
4=Convex
5=IEEE (IRIS, IBM RS/6000)
4=Convex
Select output file format            : 1
```

The file will then be converted and may be used by the application programs.

C.2 Converting Version 1 or 2 files from Cray Format

To convert a common file in Cray format to some other format, follow the following example:

```
% cfcray
Enter input  file name (<CR> to quit): cray_grid.cgd
Enter output file name               : vax_grid.cgd
1=Cray
2=VAX
4=Convex
5=IEEE (IRIS, IBM RS/6000)
Select output file format            : 2 (or 4 or 5)
```

The converted file may then be transferred to the target machine using *ftp* (binary mode), *rcp* or some other binary transfer mechanism.

C.3 Embedding CFCRAY in Application Scripts

It may be necessary to invoke CFCRAY in a batch mode (for example, on the MDC Cray). This is done by creating a data file for the CFCRAY program and then invoking it directly. This is what the interactive cfcray script under UNICOS performs.

The file format is as follows:

Line 1: *inunit*
Line 2: *infile*
Line 3: *outunit*
Line 4: *outfile*
Line 5: *output file format*

Lines 6 through the end are optional

Line 6: *cmdunit*
Line 7: *command script ...*
Line n: *command script ...*

Appendix D. Using Common Files Directly in PLOT3D

The MDA CFD project version of PLOT3D has been modified to directly accept common files as input. If a flow solver uses a common file as its grid source and produces a common file as its restart file, these files most likely can be read directly into PLOT3D. This saves time and disk space by eliminating the need to produce separate files for PLOT3D. Two additional features are available in PLOT3D when reading a common file. The `/ZONES=` "*n1-m1*, ..., *nl-ml*" qualifier to the `READ` command will read only the zones n through m, where $n < m$. The `/APPEND` qualifier to the `READ` command will append the zones in the specified common file to the end of the zones already read in to PLOT3D. *Note*: In the calculation of `IBLANK` data for particle tracing appended grids can not be interconneted. Thus all zones to be interconnected must be in the same `READ` command.

D.1 Reading a Grid File Only

To read only a grid file, the following `READ` command must be entered:

`READ/CGD=`*grid_file_name*

grid_file_name is the name of the common file (without the *.cgd* suffix) which contains the grid.

D.2 Reading Separate Grid and Flow Files

To read both a grid file and a flow file (like reading an XYZ and a Q file), issue the following `READ` command:

`READ/CGD=`*grid_file_name*`/CFL=`*flow_file_name*

grid_file_name is the name of the common file (without the *.cgd* suffix) which contains the grid. ***flow_file_name*** is the name of the common file (without the *.cfl* suffix) which contains the flowfield.

D.3 Reading a Combined Grid and Flow File

Some programs may produce a file in which the grid and flow field have been written to one file. To read such a file, issue the following `READ` command:

`READ/CGF=`*combined_file_name*

combined_file_name is the name of the common file (without the *.cgf* suffix) which contains the grid and flowfield information.

D.4 Reading Using the `/APPEND` and `/ZONES` Qualifiers

This example reads zones 4–9 and 12 from one file, and appends zone 3 from another file.

```
READ/ZONES='4-9,12-12'/CGD=GRID1/CFL=SOL1
READ/ZONES=3-3/APPEND/CGD=GRID2/CFL=SOL2
```

D.5 Dimensionalization Issues

A standand PLOT3D Q file contains data that has been non-dimensionalized. When a user requestes a plot of "dimensional" quantities (100 (density), 110 (pressure), etc.), what is really displayed is a dimensional form using predefined conditions not related to the actual conditions. For example, the normalized pressure at infinity would be displayed as $1/\gamma$.

When a common file flow file is processed, data will be displayed in SI units for dimensional plots (100 (density), 110 (pressure), etc.).

The normalized counterparts of the dimensional plots (101 (density), 111 (pressure), etc.) will display the data normalized by the infinity conditions. Therefore, the normalized pressure at infinity will now be displayed as 1, rather than $1/\gamma$ as in the case of using a Q file.

By using the common file, one can now see both the dimensional (albeit, SI values) or non-dimensional values of dimensional quantities.

D.6 Tricks

Occasionally one may want to read only the grid out of a combined grid and flow file or may want to use only the flow field out of a combined grid and flow file and use a grid from another file. To accomplish this feat, note that PLOT3D simply assumes that the file extension defines the type of data to look for. So a file with a *.cgd* suffix is assumed to contain the variables x, y, and z. Thus to cause one type of file to be treated as another type of file, simply rename it before invoking PLOT3D! This works because the basic structure of all common files is the same. It is the variables within the file that determine what it is used for. PLOT3D will only read geometry when `CGD=` is used and will ignore any other data. In a similar manner, when `CFL=` is used it wants flow field data so it ignores any geometry data that may be present.